An Accurate Time-management Unit for Real-time Processors[*]

Krishnan K. Kailas
Department of Electrical Engineering

Ashok K. Agrawala
Institute for Advanced Computer Studies
Department of Computer Science

University of Maryland
College Park, MD 20742, USA
{*krish, agrawala*}*@cs.umd.edu*

Technical Report

Abstract

Time management is an important aspect of real-time computation. Traditional high performance processors provide little or no support for management of time. In this report, we propose a time-management unit which can greatly help improve the performance of a real-time system. The proposed unit can be added to any processor architecture without affecting its performance. We also explain how the unit helps to solve the clock synchronization problems in a distributed real-time network.

[*]This work is supported in part by ONR and ARPA under contract N66001-95-C-8619 to the Computer Science Department at the University of Maryland. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency, ONR or the U.S. Government.

1 Introduction

Accurate time management functions are required for scheduling real-time tasks to meet their deadlines. Time-based scheduling techniques [15] make use of the worst-case execution time estimates of the tasks to generate deterministic schedules for hard real-time systems. With the advent of fast processors which can execute millions of instructions per second, considerable amount of computations can be done in a very short period of time. This in turn, demands accurate timing mechanisms for scheduling real-time tasks to achieve better processor utilization. A fast and accurate time keeping mechanism, such as a system clock with fine granularity is essential to implement such precise time-based scheduling algorithms. Fast internal clock of the modern processors can be made use of to implement a system clock with fine granularity. However, the support provided by the traditional high performance processor architectures are not often useful to implement these ideas even though the hardware may support a fast system clock. Some of the embedded microprocessors such as Intel 80x196, Intel386 EX, and commercial high performance processors such as Pentium and Pentium Pro [7] provide on-chip timers driven by the processor clock to implement system clocks with better time granularity. Computers that do not use such processors usually use an external hardware timer [6], [3] on the processor bus. But, these timers have been known to "lose time" during operation [19]. For example, in order to set a new value for the timer, usually a register has to be updated. But certain operations such as DMA and high priority interrupts can preempt time management functions. This can cause a delay in updating the timer register and make the system clock to lose time.

By moving all the time management functions into the processor, one can alleviate the above mentioned problems and provide a more flexible solution for the management of time. We believe that the time management functions should be made an essential feature of the processors used in real-time systems. In this report, we propose a hardware architecture of an accurate time management unit for such processors. The following are the basic functionality required for such an accurate time management unit.

- A mechanism to implement a monotonic clock. The monotonicity of clock is very important because distributed applications assumes that time-stamps produced by a clock always increases monotonically [11].
- An automatic mechanism to register the time of occurrence of user-defined events. This property is essential to accurately time-stamp events in a real-time system.
- A deterministic and atomic mechanism to read and update the system time.
- A hardware register to hold system time to the required resolution and a mechanism to increment the system time without any software intervention.
- A mechanism to compensate for the drift (internal and external) to maintain a consistent global time.

A fast and accurate time keeping mechanism can also be of help in implementing robust real-time distributed computing systems. Solutions to several design problems in distributed computing can be simplified if a global time base is available in the system [12]. Maintaining a consistent global notion of time in a distributed computing environment involves synchronizing all the local clocks. Clock synchronization problem has been studied extensively in the past and several solutions have been proposed [9], [16], [5], [13], [14], [1]. But, most of these solutions either depend on special purpose hardware or complicated protocols. This will add extra processing overhead and complexity to the system, there by increasing the clock skews in a distributed system. The time management unit architecture we have proposed here is designed taking these aspects into consideration and can be used to achieve very accurate synchronization of local clocks with little overhead.

Distributed real-time systems such as process control and data acquisition systems often demand an additional capability to accurately time stamp data and events. For example, in a distributed process control system, in order to analyze and take appropriate corrective actions to deal with "alarm conditions", knowledge about the exact sequence of events that caused the alarm condition and their time of occurrence is essential. Hence, in order to resolve the precedence of such close events, a time-stamping mechanism with fine granularity is essential. The availability of a fine granularity clock and the support provided for accurate time-stamping in the proposed time management unit can be used for identifying and time-stamping the events with better resolution.

The rest of this report is organized as follows. In section 2, drawbacks of the traditional time management techniques employed in real-time systems is explained. The architecture of the proposed time management unit is described in section 3. We explain how our solution helps to solve the clock synchronization problem in section 4. In section 5, a review of the related work done by other researchers is given.

2 Time management in Real-Time Systems

The heart of a real-time system is the scheduler that makes important decisions such as, which one of the ready tasks is to be scheduled next, when and how long each of these tasks are to be executed. The performance of such a real-time scheduler depends on it's capability to ensure that the real-time deadlines of the system are always satisfied, and to achieve high processor utilization at the same time. Most of the schedulers make use of some type of time-out mechanism to preempt the currently executing task and schedule the next task in the ready list, based on the scheduling algorithm used. Clearly, this timing mechanism should be very accurate, otherwise the tasks will be scheduled to run early or late and/or the task will be executed for longer or shorter amount of time than required. In other words, the correctness of the scheduling scheme directly depends on the timing mechanism used to implement it. Often the same timing mechanism is used to maintain the local (time-of-day or -year) clock of the system.

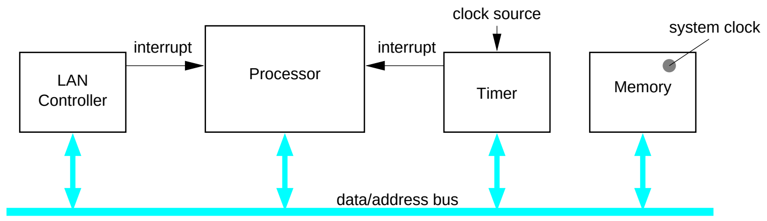

Figure 1: A node of a typical distributed real-time system

A typical node of a distributed real-time system makes use of an external timer chip as shown in figure-1 for time management. The most common timing technique is based on a system clock (software counter) in the memory updated by the *timer-tick* interrupts generated by a fixed interval timer [17, 2]. The main disadvantage of this method is that the coarse granularity of the system clock, the software timing mechanism, is limited to the order of milliseconds. This limitation arises because of the overheads associated with the *timer-tick* interrupt processing, which involves saving and restoring the processor registers, updating the

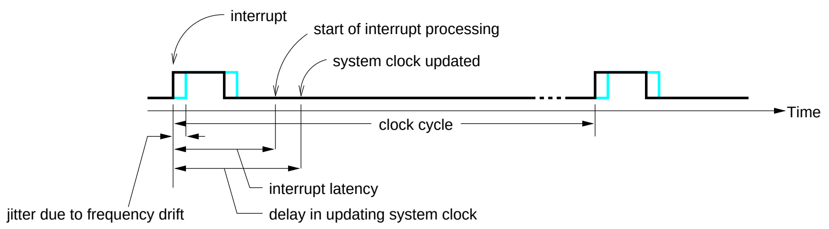
interrupt
start of interrupt processing
system clock updated
Time
clock cycle
interrupt latency
delay in updating system clock
jitter due to frequency drift

Figure 2: Timing diagram of external timer-based time management

system clock and checking the ready task list. Another disadvantage with this approach is that there is a possibility of missing the timer interrupts if the interrupts were disabled or the interrupt processing gets delayed due to operations such as DMA. The main factors that constitute the delay in updating the system clock are the following:

1. jitter in the timer interrupt signal due to frequency drifts,
2. the interrupt latency of the processor, and
3. the execution time of the interrupt service routine.

The figure-2 shows the timer interrupt signal and these delays in a typical external timer-based system. Moreover, on modern high performance processor architectures, the interrupt latency and the execution time of the interrupt service routine itself is often hard to predict[8].

An alternative to this approach is to use a programmable interval timer, which is commonly used for scheduling tasks in time-based real-time operating systems [15]. In this approach, an interval timer is loaded at each task scheduling instance with a new interval equal to the task's execution time-slot duration. Timer generates an interrupt at the end of the interval to invoke the scheduler again to schedule the next task from the ready list. The main advantage of this method over the *timer-tick* approach is the better time granularity of the scheduling clock, because the granularity depends only upon the frequency of the clock signal used to drive the timer and width of the timer register. However, this scheme also has similar disadvantage of losing the time between the initiation of the timer interrupt service routine and reloading of timer register with new interval, due to DMA operations or higher priority interrupt processing. A possible solution to this problem is to modify the timer to allow the new interval value to be added to the current count-down register contents [19]. Another solution is to use a second register to automatically load the next interval value to the timer register, as in the VAX-11 computer systems [18]. In addition to the errors that can occur in the time keeping mechanisms as mentioned above, the basic source used to drive the timer itself can generate errors due to drift. Even though the add-timer and second register method apparently solve the problem of updating the system clock for scheduling purposes, these solutions do not address problems such as providing the current time to the applications at any instant for time-stamping purposes and compensating for the drifts.

It is clear from the above discussion that the existing time management approaches are not satisfactory in performance to provide the fine time granularity, accuracy and flexibility demanded by the real-time computing systems of today. The problem with the existing solutions is that they address and solve the problems separately, resulting in solutions that are not comprehensive. Hence these solutions, as discussed above, often fail to provide guaranteed error-free performance always. We strongly believe that in order to provide a comprehensive solution to meet all the above requirements, the solution should be based on hardware, and must be implemented within the processor. Moving the time management functions to the processor level has several advantages which can not be achieved otherwise. For example, with the traditional *timer-tick* approach, the resolution of time measurement is limited to the coarse granularity of the *timer-tick*s. The granularity of the proposed hardware time management unit, as explained later, is much higher than that can be achieved with *timer-tick* approach. The time management unit also set free the processor from interacting with external hardware to implement time management functions and there by providing more bus bandwidth and computing resources to other tasks. Most of the time management functions can be made independent of other CPU activities, and thus they can be carried out in parallel. This can provide substantial improvements in the performance of time management functions and can be used of to implement accurate clock synchronization algorithms and other distributed applications.

3 Time Management Unit Architecture

The proposed time-management unit works in parallel with the processor, without using any of the computational resources of the CPU. This allows the time-management unit to provide very accurate time management functionality without affecting the performance of the processor. The architecture of the time management unit is shown in figure-3. It consists of a set of registers accessible to the CPU, a *Limit* register and associated logic. A drift-free clock signal is derived from the clock source using a *Rate Adjustment Unit*. The clock source can be the internal clock used in the processor itself or a stable external clock source such as a crystal oscillator. The *Rate Adjustment Unit* consists of a frequency divider (counter) for scaling down the input clock source frequency and a phase adjustment counter to apply corrections for small changes in frequency. The *Rate Adjustment Unit* makes necessary corrections to nullify any changes in the frequency of the clock source due to drift (see the next section for a detailed description about the functioning of this unit).

The system time is maintained in the *Physical Time* register, which is a 64-bit counter incremented at a rate specified by the output clock frequency of the *Rate Adjustment Unit*. The granularity of time maintained by the system is therefore defined by the output clock frequency of the *Rate Adjustment Unit*. For example, a 64-bit register can represent a time span of millions of years and provide a time granularity of the order of 1/10th of micro seconds with a 10 MHz clock derived using the *Rate Adjustment Unit*. The *Physical Time* register can be accessed by system software as a CPU register to read or modify it's contents. The system time T_S is compared with a *Limit* register T_L at each processor clock cycle and an interrupt is generated when the condition $T_S \geq T_L$ is satisfied. This interrupt signal can be used for real-time task scheduling and precisely initiating time-based events. The interrupt signal will be reset only when the *Limit* register is modified.

The user accessible registers of the proposed time management unit is shown in figure- 4. All the registers except the *Rate Divisor* register and *Mode Selector* register are 64-bit registers. There are 3 modes of operation for the proposed time management unit, based on the way in which the *Limit* register content is updated. The modes can be selected by writing appropriate control words in the *Mode Selector* register. The three modes of operations is described below:

Absolute time mode: This mode may be used for scheduling tasks precisely at a given absolute time. The time at which the interrupt is to be generated is specified in the *Absolute time* register, which is accessible to the CPU. In this mode, the *Limit* register is loaded with the contents of *Absolute time* register and at each processor clock cycle it is compared with the current physical time.

ΔT mode: This mode is intented for generating precise delays by emulating a one-shot timer. The desired delay time is specified in the ΔT register. In this mode, the *Limit* register

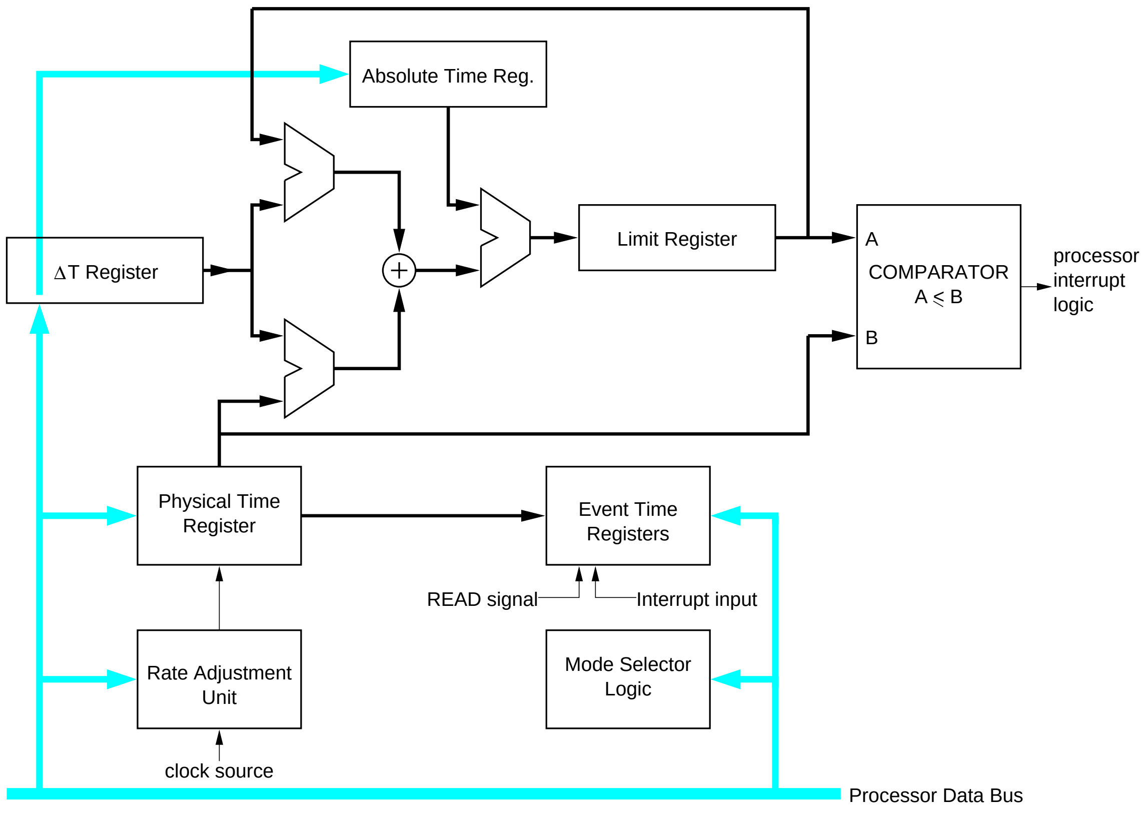

Figure 3: Time Management Unit Architecture

will be loaded with the sum of the current contents of the *Physical Time* register and ΔT register, in the next clock cycle after the the ΔT register is updated. The idea is to prevent any loss of time as in timer tick-based approach.

Auto-reload mode: In this mode, after generating the interrupt, the *Limit* register is automatically loaded with a new value similar to the ΔT mode. The new value of the *Limit* register is generated by adding the current contents of *Limit* register and the ΔT register. For example, if T_S is the physical time (the contents of the *Limit* register) at the instant when the interrupt is generated, and ΔT is the delay time, then the comparator will generate the next interrupt after $T_S + \Delta T$ seconds. The main difference between this mode and the ΔT mode is that in this mode the changes in ΔT register will be effective in the cycle after the interrupt. The idea is to eliminate the delays (refer to figure-2) that would have occurred in the ΔT mode if the same functionality is implemented making use of the timer interrupt service routine to load a new time interval after each interrupt. This mode can be used to implement very accurate time-based schedulers such as the one used in the Maruti hard real-time operating system [15].

The proposed time-management unit also provides an accurate mechanism to time-stamp events that occur in the system. This is made feasible because, at any processor cycle using a single register transfer instruction, the current physical time can be read into one of the *Event Time* registers. The same operation can also be initiated by an external interrupt signal. This

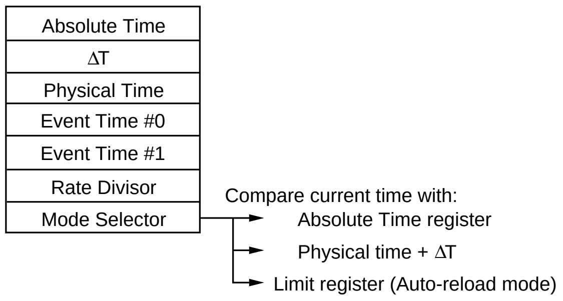

Figure 4: Time Management Unit Registers

facilitates accurate time-stamping of external events without interfering the CPU computations. In the next section, we explain how to make use of this feature to implement accurate clock synchronization algorithms.

4 Support for Clock Synchronization

The essential requirements of a clock synchronization algorithm for distributed real-time systems can be found in [9, 19]. A common characteristic of all clock synchronization algorithms is that each node computes periodically the deviation of its local clock from a global time base [16]. These clock synchronization algorithms make use of the knowledge about the local clocks of other nodes or a master node in the system to compute the corrections to the local clock. Time stamped packets are used for exchanging the current time of the local clocks in the system. But, in a distributed system, there is a large variability in the time taken by a packet from the instant it is submitted for transmission at the sender node to the time it is processed at the receiver node. This jitter associated with the message passing may be attributed to the dynamics of the network and the processing delays at the sender and destination nodes. For example, the ethernet protocol can introduce certain amount of uncertainty which increases with the network traffic [10]. However, by choosing protocols such as TDMA to pre-allocate slots for time message packets, the variability due to the network dynamics can be bounded [4, 15]. However, the jitter persists due to the unpredictable processing delays at the nodes resulting from operations such as non-preemptible interrupt processing and DMA. It is clear from the above discussion that regardless of the algorithm used for clock synchronization, the accuracy of the technique is affected by time stamping operation at the sender and receiver nodes. Clearly, a mechanism is needed to accurately time-stamp the packets just before they are transmitted at the sender and as soon as they arrive at the receiver. The proposed time management unit provides such a mechanism to solve this problem by automatically time-stamping the packets. The packets are time stamped on arrival making use of the interrupt signal from the network interface card without any processor intervention. The current physical time will be latched in the *Event Time* register by the interrupt signal, which can be made use of to accurately time-stamp the arrival-time of the packets. The *Physical time* register can be read for time stamping the packet just before they are sent. Thus, with the help of the proposed time management unit, without using any external hardware, the packets can be accurately time-stamped to implement clock synchronization algorithms. Moreover, the Absolute Time mode of the proposed time management unit can be used for precisely scheduling the send time of time messages.

In addition to the errors that occur due to the jitter in the message passing, the basic clock source itself can generate errors due to drift. This drift in frequency of the clock source is due to

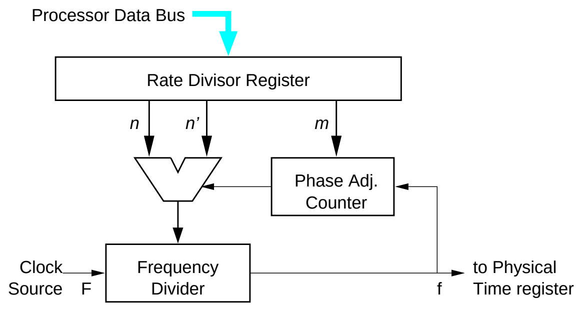

Figure 5: Rate Adjustment Unit

temperature variations and aging, and must be corrected. A discrete correction applied to the *Physical Time* register can cause the local clock to instantaneously leap forward or be set back and then run at the previous rate, thereby violating the monotonicity property. This problem can be avoided by *amortizing* (i.e., spreading out) the correction continuously over a time interval and this clock adjustment technique is called *amortization* [16]. The *Rate Adjustment unit* implements this technique in hardware to avoid any abrupt *jumps* in the local clock.

A frequency divider making use of a simple binary counter may not be sufficient to derive the desired output frequency accurately from the clock source. This is because of the truncation errors in the approximation of the scaling factor to an integer value. However, it is possible to derive fairly accurate output clock signal by changing the width of a few clock pulses out of a fixed number of clock pulses periodically, so that the average frequency of the output signal is very close to the required value. The *Rate Adjustment unit* makes use of this technique to derive the desired clock frequency. The unit consists of a frequency divider (counter) for scaling down the input clock source frequency and a phase adjustment counter[1] to apply phase corrections as shown in figure-5. Very small changes in output frequency is taken care of by re-loading the frequency divider with a slightly higher or lower count than the normal count at a rate specified by the phase adjustment counter. If F is the frequency of the clock source and f is the desired output clock frequency, then the normal scaling factor of the frequency divider is given by $n = \lceil F/f \rceil$. If F/f is not an integer, then one out of every $1/m$ output clock cycles, the frequency divider is loaded with a modified scaling factor $n' = n \pm k$, where k is an integer (note that $k = 0$ if F/f is an integer). The phase adjustment rate m, may be computed using the following relationship.

$$(1-m)n + mn' = \frac{F}{f}$$

Most of the time, at the time of re-synchronization, only the phase adjustment count, n' and the phase adjustment rate, m, needs to be updated. The parameters n, n' and m are made accessible to the software at the *Rate Divisor* register.

5 Related Work

The importance of time management in real-time systems has been identified by researchers for quite some time. The problems with the interval based timing mechanisms and the lack of coordination between the hardware timers and software were discussed by Volz and Mudge

[1]The counter is called the *phase* adjustment counter because the counter changes the width or the phase of the output signal by a small amount.

in [19]. They suggested the use of absolute time as a solution and proposed an instruction level timing mechanism to accomplish this. They have also mentioned the idea of placing the timing functions on the CPU chip for scheduling applications. In contrast, the architecture we have proposed here is more versatile and generic in nature – the support for instruction-level scheduling is one of the features supported by the proposed architecture. The architecture we have proposed does not explicitly specify the format in which the time is represented, though the architecture supports the absolute time representation mentioned in [19, 20]. The reason for making it a generic architecture is to make it easy to adapt with minimal modifications to the existing applications and to easily adapt to new time representation formats that may come up in the future.

The Mars project [4, 9] make use of proprietary network interface logic based on a clock synchronization unit chip, to automatically generate time-stamps. The scheme they have proposed provides most of the functionality required by a time management unit. However, the portability of their solution to other platforms is highly restricted because it is based on external hardware. Whereas our solution is transparent to most of the hardware and the network interface logic used in the system, there by making it easily portable to other platforms. The basic idea of the rate adjustment scheme for deriving clock signal proposed in this report is similar to the adjustable rate clock proposed by Volz *et al.* for clock synchronization in IEEE 896 Futurebus+ systems [20]. The differences are mainly of hardware implementation details.

Another hardware-based clock synchronization technique for synchronizing the clock signals can be found in [1]. This technique implements a modified version of the interactive convergence algorithm CNV [13] and assumes that the actual clock signals are available for skew measurements. However, this scheme is not suitable for large distributed system because of the problems associated with distributing the clock signal over large distances.

The hardware-assisted software clock synchronization scheme proposed by Ramanathan *et al.*[14] make use of an algorithm similar to CNV for a distributed system with point-to-point interconnection topology. They emphasize on the algorithmic aspect than the implementation of the hardware support required. The resolution of their technique for applying corrections to the logical clock at nodes is limited by the frequency of the clock source and the scaling factor of their scheme can only assume one fixed integer value. As a result of which their scheme can not compensate for very small variations in frequency. Moreover, the architecture proposed here is aimed at providing mechanisms for efficient implementation of distributed clock synchronization algorithms with minimum software overheads and better accuracy, rather than implementing a specific algorithm in hardware.

6 Conclusion

In this report we have proposed an accurate time-management unit architecture for real-time processors. Our design is motivated by the lack of support provided for time management in the modern processors. The proposed time management unit can be incorporated into any processor architecture with little extra logic. With the recent developments in VLSI technology, such a time management unit with nanoseconds resolution can be easily implemented. The basic idea behind the proposed time management unit is to exploit the parallelism between the time management functions and normal computing operations of the processor, at the same time providing an instruction-level mechanism to access the system time. We believe that moving the time management functionality into the processor will greatly help to generate better solutions in terms of performance, simplicity and maintainability.

7 Future Work

In order to support the proposed on-chip time-management unit, the processor architecture must provide a deterministic instruction-level mechanism to interact with the time-management unit hardware. In the modern multiple-issue processors supporting out-of-order execution of instructions, it is hard to predict the delay between instruction issue and retirement. Therefore, on such processors a special instruction for reading the current physical time into the event time register is not sufficient, unless there is a mechanism to ensure that the instruction can be executed within a deterministic time interval. We would like to address this problem in our future research. At present, as a first step towards understanding the problem, we are looking at the timing issues and temporal accuracy of one of the commercial off the shelf modern processors.

Acknowledgment

Intel386 EX, Pentium and Pentium Pro are registered trademarks of Intel Corporation.

References

[1] Y. Baek, H-K. Lee, and H. Yoon. New hardware-based clock synchronization for the Byzatine fault. *Electronics Letters*, 28(21):2018–2019, October 1992.

[2] Dipto Chakravarty. *POWER RISC System/6000: Concepts, facilities, and architecture*, chapter 14. McGraw-Hill, Inc., New York, 1994.

[3] *Product Data book*, chapter 6. Dallas Semiconductor Corporation, Dallas, TX, 1992-93.

[4] Hermann Kopetz et al. Distributed Fault-tolerant Real-Time Systems: The Mars Approach. *IEEE Micro*, pages 25–40, February 1989.

[5] Flaviu Cristian. Probabilistic Approach to Distributed Clock Synchronization. In *9th International Conference on Distributed Computing Systems*, pages 288–296. IEEE, 1989.

[6] *Microprocessor and Peripheral Handbook*, volume II, chapter 6. Intel Corporation, Santa Clara, CA, 1989.

[7] *Pentium Pro Family Developer's Manual*, volume 1-3. Intel Corporation, Mt. Prospect, IL, 1996.

[8] P. Koopman. Perils of the PC Cache. *Embedded Systems Programming*, 6(5):26–34, May 1993.

[9] Hermann Kopetz and Wilhelm Ochsenreiter. Clock Synchronization in Distributed Real-Time Systems. *IEEE Transactions on Computers*, C-36(8):933–940, August 1987.

[10] James F. Kurose, Mischa Schwartz, and Yechiam Yemini. Multiple-Access Protocols and Time-Constrainted Communication. *ACM Computing Surveys*, 16(1):43–70, March 1984.

[11] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed system. *Communications of ACM*, 21(7):558–565, July 1978.

[12] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages Syst.*, 6(2):254–280, April 1984.

[13] L. Lamport and P.M. Meilliar-Smith. Synchronizing Clocks in the presence of Faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[14] P. Ramanathan, Dilip D. Kandalur, and Kang G. Shin. Hardware-Assisted Software Clock Synchronization for Homogeneous Distributed Systems. *IEEE Transactions on Computers*, 39(4):514–524, April 1990.

[15] M. Saksena, J. da Silva, and Ashok K. Agrawala. Design and Implementation of Maruti-II. In Sang H. Son, editor, *Principles of Real-Time Systems*. Prentice Hall, Englewood Cliffs, N.J., 1995. Also available as University of Maryland CS Tech Report CS-TR-3181.

[16] Frank Schmuck and Flaviu Cristian. Continuous clock amortization need not affect the precision of a clock synchronization algorithm. Technical Report RJ 7290 (68547), IBM Almaden Research Center, San Jose, CA, 1990.

[17] Andrew S. Tanenbaum. *Modern Operating Systems*, chapter 5. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992.

[18] *VAX hardware handbook*, chapter 20. Digital Equipment Corporation, Maynard, Mass., 1982.

[19] Richard A. Volz and Trevor N. Mudge. Instruction Level Timing Mechanism for Accurate Real-Time Task Scheduling. *ACM Transactions on Computers*, C-36(8):988–993, August 1987.

[20] Richard A. Volz, Lui Sha, and Dwight Wilcox. Maintaining Global Time in Futurebus+. *The Journal of Real-Time Systems*, 3:5–17, 1991.